# Lab 11
## Platform Device Driver

EELE 467
**Due date: 12/02/2024**

---

In this lab, we'll create a device driver for our LED patterns component. We're going to create the device driver from scratch, building it up bit-by-bit and learning about each and every piece along the way.

## Introduction to Device Drivers

A good overview of the types of device drivers found in Linux can be seen in figure 1, which is from the (free) book *Linux Device Drivers* found at https://lwn.net/Kernel/LDD3/.

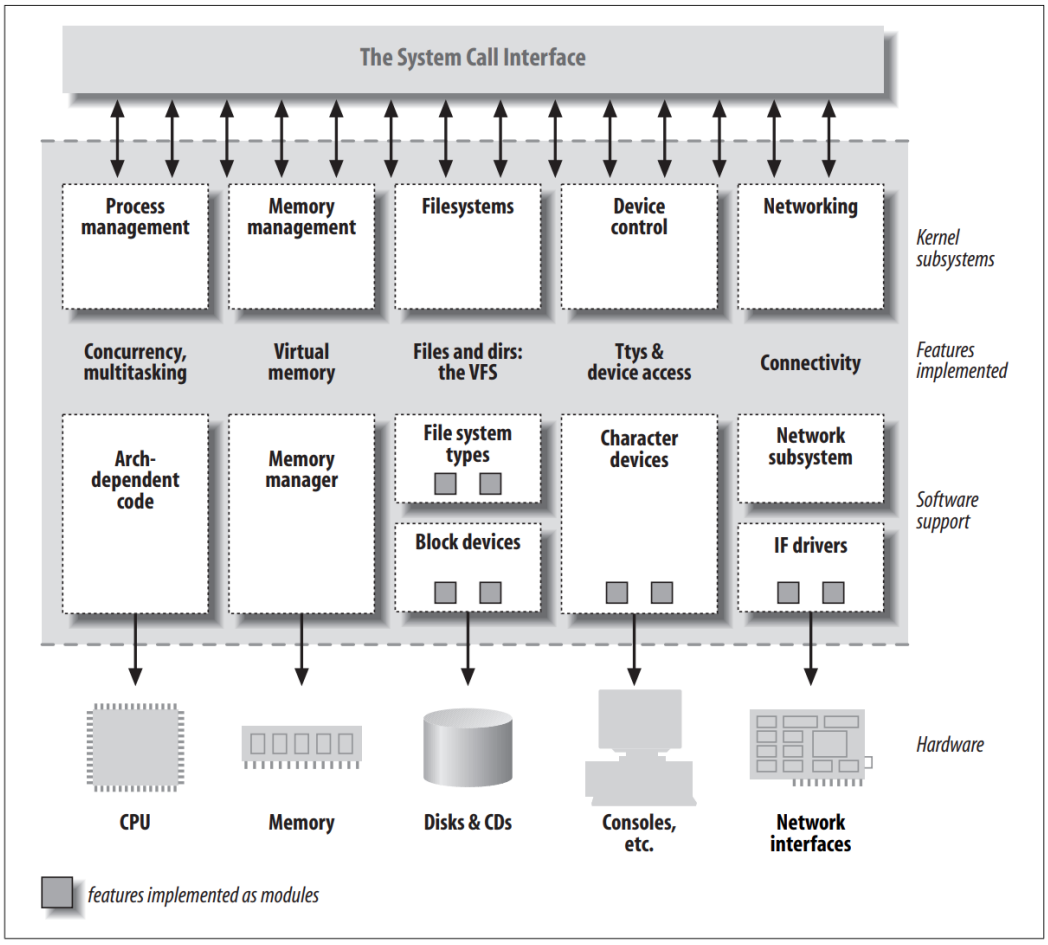


*Figure 1-1. A split view of the kernel*

Figure 1: **Types of Devices and Drivers found in Linux**. The three main classes of device drivers are *block devices*, *character devices*, and *network interfaces*.

**Network interfaces** are a special category that deal with the transmission and receipt of data packets from other computers.

**Block devices** are closely related to file systems where data movement is more efficient when the data is moved in blocks of 512 bytes or larger (and a power of two). Typically there is a memory buffer for storing blocks and the data can be accessed in random order (e.g. file seek).

**Character devices** involve transactions of single bytes (or single 32-bit words in our case) that are sequentially accessed (a stream of bytes read from a keyboard or sent to a serial port). Since we will be reading and writing single 32-bit data words to specific memory locations, the device driver type that we will implement will be a *character device driver* in order to control our custom device (i.e. reading and writing 32-bit words from/to our custom registers in our custom dataplane component in Platform Designer).

## Buses

Desktop computers have two convenient buses, the *Universal Serial Bus* (**USB**) and the *Peripheral Component Interconnect* (**PCI** or **PCIe**) buses (e stands for express). Both of these buses allow hardware to be attached to the computer without having to inform the operating system (OS) ahead of time. These buses allow hardware to be **discoverable** where the hardware can be plugged in and the hardware says to the OS "Here I am" and tells the OS what device they are and what resources they have. The OS can then deal with them as they appear or disappear.

However, many times in embedded systems, there is **hardware that is not discoverable**. The OS needs to be informed that these hardware devices exist and what resources are available for them. These devices are regarded as connected to a *virtual bus*, which is called the **platform bus**. Linux device drivers that are created for these devices are known as **platform drivers**. Since our LED patterns component in Platform Designer is not discoverable, we will treat it has a platform device and create a platform driver for it.

In addition to the platform bus, the i2c and spi physical buses are commonly found in embedded systems. These devices aren't hot-pluggable and have to be defined statically, e.g. in the device tree. The kernel has subsystems for these busses, amongst many others that you might encounter in your lifetime as an embedded systems engineer.

## Subsystems

The kernel has a large number of subsystems that cover a large range of device types, e.g. input, leds, network, audio, etc. These subsystems make writing drivers easier because they contain common functionality that all drivers of a particular type would need or want; they also help provide a consistent API to user-space, e.g. all keyboards will expose the same interfaces. Figure 2 shows some subsystems that utilize the character driver API.

While there are many subsystems in the kernel, some devices, e.g. custom devices in the FPGA fabric, don't fit cleanly into any of the subsystems. The **misc subsystem** was created to handle devices that don't cleanly fit into an existing subsystem. If the driver can make use of the *character driver* API and could be implemented as a *raw character driver*, the misc subsystem layers on top and makes the work of developing a character driver easier. We'll be using the misc subsystem in our driver.

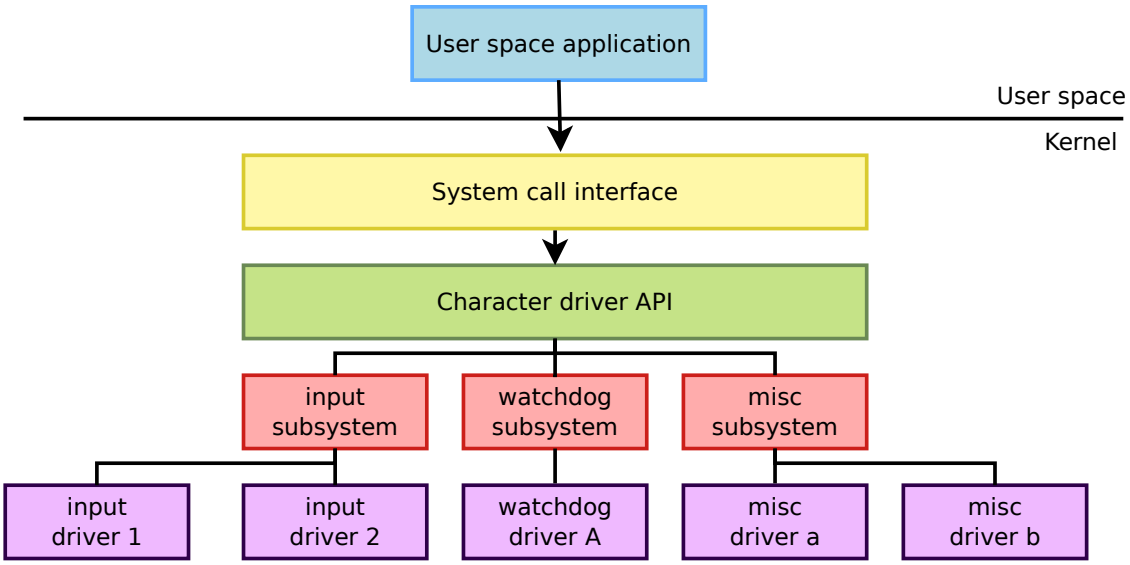


Figure 2: Diagram of different subsystems that use the character driver API. This diagram was created by engineers at Bootlin and is licensed under CC BY-SA 3.0.

## Creating the Platform Device Driver

In order to better understand the driver, we will build it piece-by-piece. You'll have to make platform drivers for your final project, so pay attention! 🙂

### Probing and Removing a Platform Device

We'll start by exploring how our driver inserts and removes platform devices. This exploration will be at a somewhat superficial level, since most of the details are handled by the kernel's driver core.

When the kernel is made aware of a device, it first has to find what driver(s) are compatible with the device. In our case, the device tree subsystem is what informs the kernel that our device exists. **If the `compatible` property in a device tree node matches the compatible string that we define in our driver, then the device gets bound to our driver.** Once a device has been bound to the driver, the driver's `probe` function gets called; similarly, when a device is removed from the system, the driver's `remove` function is called.

Before we can start probing and removing devices, we need to define our platform driver. Listing 1 shows the `plaform_driver` `struct` definition, and listing 2 shows what our platform driver instance will look like. We need to provide *function pointers* to our `probe` and `remove` functions, as well as the driver owner, name, and device tree match table (`of_match_table`)[1]. We also need to set up some module documentation macros; in particular, we need to use a GPL-compatible license because some of the platform driver code we use is exported with `EXPORT_SYMBOL_GPL()`, essentially meaning that our driver is a derived work of the kernel and must be licensed under a GPL-compatible license.

[1] `of` stands for Open Firmware, which is the standard that originally defined the device tree; see this page for a brief history

```c
struct platform_driver {
    int (*probe)(struct platform_device *);
    int (*remove)(struct platform_device *);
    void (*shutdown)(struct platform_device *);
    int (*suspend)(struct platform_device *, pm_message_t state);
    int (*resume)(struct platform_device *);
    struct device_driver driver;
    const struct platform_device_id *id_table;
    bool prevent_deferred_probe;
};
```

Listing 1: `platform_driver` **`struct`** definition from include/linux/platform_device.h. For our purposes, we only need to define the probe/remove function pointers and some fields in the `device_driver` **`struct`**, as shown in Listing 2. Look at the `device_driver` **`struct`** documentation here: include/linux/device/driver.h

```c
/*
 * struct led_patterns_driver - Platform driver struct for the led_patterns driver
 * @probe: Function that's called when a device is found
 * @remove: Function that's called when a device is removed
 * @driver.owner: Which module owns this driver
 * @driver.name: Name of the led_patterns driver
 * @driver.of_match_table: Device tree match table
 */
static struct platform_driver led_patterns_driver = {
    .probe = led_patterns_probe,
    .remove = led_patterns_remove,
    .driver = {
        .owner = THIS_MODULE,
        .name = "led_patterns",
        .of_match_table = led_patterns_of_match,
    },
};

/*
 * We don't need to do anything special in module init/exit.
 * This macro automatically handles module init/exit.
 */
module_platform_driver(led_patterns_driver);

MODULE_LICENSE("Dual MIT/GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("led_patterns driver");
```

Listing 2: Our LED patterns platform driver **`struct`**. The `probe` and `remove` fields are function pointers to our probe and remove functions (defined in Listing 3). The `owner` field specifies which module owns the driver. `of_match_table` is what gets used to match device tree nodes to our driver. The `module_platform_driver()` macro creates the module init/exit code for us.

For now, we'll create trivial probe and remove functions that just let us know the function has been entered. Listing 3 shows the probe and remove functions you need to implement. You can get the *address* of a function by using the function name without parentheses, which is what is done in Listing 2.

```
/**
 * led_patterns_probe() - Initialize device when a match is found
 * @pdev: Platform device structure associated with our led patterns device;
 *        pdev is automatically created by the driver core based upon our
 *        led patterns device tree node.
 *
 * When a device that is compatible with this led patterns driver is found, the
 * driver's probe function is called. This probe function gets called by the
 * kernel when an led_patterns device is found in the device tree.
 */
static int led_patterns_probe(struct platform_device *pdev)
{
    pr_info("led_patterns_probe\n");

    return 0;
}

/**
 * led_patterns_probe() - Remove an led patterns device.
 * @pdev: Platform device structure associated with our led patterns device.
 *
 * This function is called when an led patterns devicee is removed or
 * the driver is removed.
 */
static int led_patterns_remove(struct platform_device *pdev)
{
    pr_info("led_patterns_remove\n");

    return 0;
}
```

Listing 3: Trivial probe and remove functions.

In order for our driver to be matched with compatible devices, we need to create an array of of_device_id **struct**s, as shown in listing 4. The of_device_id array should be **static** (not accessible by other files, i.e. private) and **const** (code shouldn't be able to modify the compatible string at run-time). **You need to set the compatible string to** "lastname,led_patterns", **where** "lastname" **is your last name**, i.e., replace "adsd" with your last name. Typically, this would be an abbreviated string of the device manufacturer's name.

```c
/*
 * Define the compatible property used for matching devices to this driver,
 * then add our device id structure to the kernel's device table. For a device
 * to be matched with this driver, its device tree node must use the same
 * compatible string as defined here.
 */
static const struct of_device_id led_patterns_of_match[] = {
    { .compatible = "adsd,led_patterns", },
    { }
};
MODULE_DEVICE_TABLE(of, led_patterns_of_match);
```

Listing 4: This code defines what devices the driver is compatible with and adds our module to the kernel's device table. `struct of_device_id` is defined in include/linux/mod_devicetable.h. We only need to set the `compatible` field, which is what's used to match against the `compatible` property in the device tree. `compatible` is of the form "manufacturer,device". `MODULE_DEVICE_TABLE` allows the kernel to load our module if a compatible device is found; in our case, this won't happen because our module is out-of-tree.

**Create the driver by following these steps**:

1. Add the code in Listings 2, 3, and 4 to a file named `led_patterns.c` in your repository's `linux/led-patterns/` directory. The `probe` and `remove` functions need to be declared before the `led_patterns_driver` struct, unless you define function prototypes for the `probe` and `remove` functions.
2. Add the following headers:
    - `#include <linux/module.h>`: basic kernel module definitions
    - `#include <linux/platform_device.h>`: platform driver/device definitions
    - `#include <linux/mod_devicetable.h>`: `of_device_id`, `MODULE_DEVICE_TABLE`
3. Compile your `led_patterns` module. Copy and edit the Makefile you made in HW 8 for your "hello world" module.

The order in which you define all the functions matters. Each function, variable, etc. needs to be defined before it is used.

### Verification

1. Copy `led_patterns.ko` to `/srv/nfs/de10nano/ubuntu-rootfs/home/soc`.
2. In your serial console (e.g. PuTTY), load your `led_patterns` module using `sudo insmod`.
3. Verify that your driver is loaded by running `lsmod`.
4. Look for your driver's print statements by typing `dmesg | tail`.

What happened when loaded your module? Did your driver print "led_patterns_probe"? It shouldn't have because Linux doesn't know that an `led_patterns` device exists; we need to add our `led_patterns` device to the device tree!

### Modifying the Device Tree

When we create new hardware (i.e., our custom component in Platform Designer), we need to inform Linux that this hardware exists. We do this by creating a new node in the *device tree*.

We need to add a new device tree node for the `led_patterns` component. This will be a node contained inside the `base_fpga_region` node of the device tree. The syntax of the node will be:

```
label: node-name@base_address {
    compatible = "manufacturer,model";
    reg = <base_address span>;
};
```

We'll define our `led_patterns` node at the root of the device tree, as shown in listing 5. (I'm pretty sure we should define our device inside the base-fpga-region node, but that's not working...)

```
#include "socfpga_cyclone5_de10nano.dtsi"

/{
    led_patterns: led_patterns@ff200020 {
        compatible = "adsd,led_patterns";
        reg = <0xff200020 16>;
    };
};
```

Listing 5: `socfpga_cyclone5_de10nano_led_patterns.dts` device tree. We add our node at the root of the device tree. Assuming our component has a base address of 0x20 in Platform Designer, the physical address is 0xff200020 (the bus' base address is 0xff200000, which we add 0x20 to). The span is the number of bytes the component's memory space uses; that is, 4 times the *maximum* number of registers the component could have.

**Step 1** Add the `led_patterns` node as shown in listing 5 to your `socfpga_cyclone5_de10nano_led_patterns.dts` file.

**Step 2** Change the **`compatible`** property to `"lastname,led_patterns"` to match what you did in your driver code.

Ensure that `led_patterns`'s base address and span match what has been assigned by Platform Designer.

The compatible strings need to match *exactly*. Otherwise, the driver will not get bound to the device.

**Step 3** Compile your device tree and copy the dtb to your TFTP server.

**Verification**

Reboot your SoC FPGA and load your `led_patterns` driver. When you run `dmesg | tail`, you should now see "led_patterns_probe" printed; similarly, you should see "led_patterns_remove" printed when you remove from your driver. That was fun, wasn't it? 🥳🤓

### Setting up a State Container and Accessing Device Memory

Printing messages is fun and all, but we actually want to use our platform device. In this section, we'll take some baby steps towards that goal: we'll set up a state container for the led patterns, get access to the hardware's memory, and turn the LEDs on/off in the probe/remove functions.

The state container design pattern is common in device drivers. Essentially, the state container is a **`struct`** that *contains* the *state* of our device, e.g., addresses, values, etc.—anything we want to associate with the device and keep track of. One of the primary reasons for this design pattern is that drivers assume they will be bound to multiple devices (you could instantiate `led_patterns` multiple times in Platform Designer if you had external LEDs connected up to I/O pins), so we need a separate state for each device. The device's state is passed around to any function that needs to access or modify the state. Listing 6 shows a basic state container for our LED patterns device.

```
11 /**
12  * struct led_patterns_dev - Private led patterns device struct.
13  * @base_addr: Pointer to the component's base address
14  * @hps_led_control: Address of the hps_led_control register
15  * @base_period: Address of the base_period register
16  * @led_reg: Address of the led_reg register
17  *
18  * An led_patterns_dev struct gets created for each led patterns component.
19  */
20 struct led_patterns_dev {
21     void __iomem *base_addr;
22     void __iomem *hps_led_control;
23     void __iomem *base_period;
24     void __iomem *led_reg;
25 };
```

Listing 6: LED patterns state container. The `led_patterns_dev` **`struct`** contains pointers to each of the registers. The `__iomem` token specifies that the pointer points to I/O memory, which allows the kernel to perform some code-correctness checks[2].

In order to access and control our platform device, the probe function needs to do three primary things, as shown in Listing 7:

1. Allocate kernel-space memory for our state container (lines 47-48 in listing 7)
2. Request and remap the device's physical memory into the kernel's virtual address space (line 60 in listing 7)
3. Attach our state container to the platform device (line 78 in listing 7)

[2] See this great stackoverflow answer for more details on the `__iomem` token.

```c
static int led_patterns_probe(struct platform_device *pdev)
{
    struct led_patterns_dev *priv;

    /*
     * Allocate kernel memory for the led patterns device and set it to 0.
     * GFP_KERNEL specifies that we are allocating normal kernel RAM;
     * see the kmalloc documentation for more info. The allocated memory
     * is automatically freed when the device is removed.
     */
    priv = devm_kzalloc(&pdev->dev, sizeof(struct led_patterns_dev),
                        GFP_KERNEL);
    if (!priv) {
        pr_err("Failed to allocate memory\n");
        return -ENOMEM;
    }

    /*
     * Request and remap the device's memory region. Requesting the region
     * make sure nobody else can use that memory. The memory is remapped
     * into the kernel's virtual address space because we don't have access
     * to physical memory locations.
     */
    priv->base_addr = devm_platform_ioremap_resource(pdev, 0);
    if (IS_ERR(priv->base_addr)) {
        pr_err("Failed to request/remap platform device resource\n");
        return PTR_ERR(priv->base_addr);
    }

    // Set the memory addresses for each register.
    priv->hps_led_control = priv->base_addr + HPS_LED_CONTROL_OFFSET;
    priv->base_period = priv->base_addr + BASE_PERIOD_OFFSET;
    priv->led_reg = priv->base_addr + LED_REG_OFFSET;

    // Enable software-control mode and turn all the LEDs on, just for fun.
    iowrite32(1, priv->hps_led_control);
    iowrite32(0xff, priv->led_reg);

    /* Attach the led patterns's private data to the platform device's struct.
     * This is so we can access our state container in the other functions.
     */
    platform_set_drvdata(pdev, priv);

    pr_info("led_patterns_probe successful\n");
```

Listing 7: More useful probe function. This probe function allocates kernel memory for the led_patterns_dev state container, gets a pointer to component's base address, turns the LEDs on, and attaches our state container to the platform device's driver_data field.

In the old days (pre-2007 or so), memory and resources that were allocated in `probe()` needed to be manually released in `remove()`, e.g. for every `kmalloc()` used in the `probe` function, there needed be a corresponding `kfree()` in the `remove` function. This led to many bugs during device initialization and detach, which eventually prompted the creation of devres, which stands for (Managed) **Device Resource**. The devres documentation humorously describes the motivation as follows

> As with many other device drivers, libata low level drivers have sufficient bugs in ->remove and ->probe failure path. Well, yes, that's probably because libata low level driver developers are lazy bunch, but aren't all low level driver developers? After spending a day fiddling with braindamaged hardware with no document or braindamaged document, if it's finally working, well, it's working.
>
> So, many low level drivers end up leaking resources on driver detach and having half broken failure path implementation in ->probe() which would leak resources or even cause oops when failure occurs

The upshot of using devres is that resources managed by it are automatically released upon device detach, which eliminates a lot of bugs and makes our lives and code much simpler. devres functions are prefixed with `devm_`. devres is "basically [a] linked list of arbitrarily sized memory areas associated with a `struct device`".

`devm_kzalloc` is the first devres function we use. We use it to allocate memory for our state container. In summary, it creates a memory region of size `sizeof(struct led_patterns_dev)` (the size of our state container) and adds the corresponding pointer to the `device struct` contained in our platform device (`&pdev->dev` is a pointer to the `device struct`). See the `devm_kzalloc` and `devm_kmalloc` definitions for details.

The next thing our probe function has to do is gain access to the LED patterns component's physical memory. This is done with the `devm_platform_ioremap_resource` function. This function first requests access to the memory region, which ensures that nobody else can use that memory. The function then remaps the physical memory into the kernel's virtual memory space; this remapping is necessary because the kernel doesn't have direct access to the device's physical address space; Figure 3 shows an example of memory remapping. The output of `devm_platform_ioremap_resource` is a pointer (in virtual address space) to the component's base address.

Lines 72-73 in listing 7 enable software-control mode and turn the LEDs on. `iowrite32` takes a 32-bit value and a `void` pointer, then handles all the architecture-dependent details needed to actually write to our hardware (writing to memory-mapped I/O is different on x86 vs. ARM vs. PowerPC vs. MIPS, etc.).

The last thing our probe function needs to do is attach our state container to it's associated `device struct`. `platform_set_drvdata()` handles this for us. It takes a pointer to a platform device and a pointer to our state container, and then sets the device's `driver_data` field, i.e., `pdev->dev->driver_data = priv`. We need to attach our state container to the device because the (platform) device is what gets passed around to our functions (we've seen this already in our probe/remove functions).

Although there aren't many lines of code, there's a lot going on in the `probe` function! Fortunately, since we used devres, our `remove` function is much simpler. All we need to do is get our state container from the platform device struct so we can put the device back into hardware-control mode (this is not strictly necessary), as shown in listing 8.

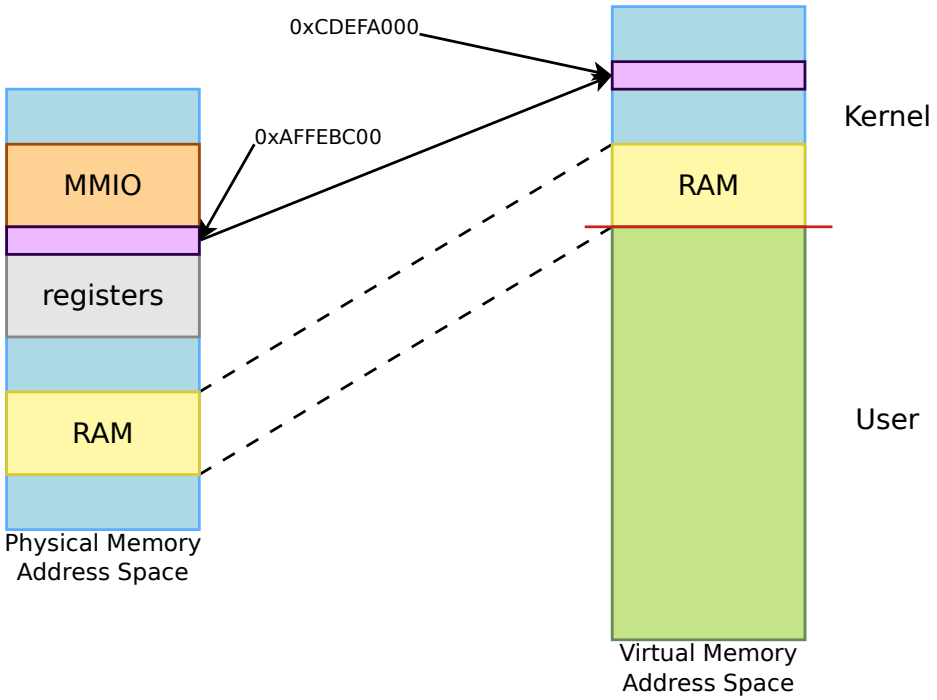


Figure 3: Example of remapping memory-mapped I/O into the kernel's virtual address space. The physical memory region's base address is 0xAFFEBC00, which gets remapped to address 0xCDEFA000 in the kernel's virtual address space. This ioremap diagram was created by Michael Opdenacker from Bootlin and is licensed under CC BY-SA 3.0.

```
92  static int led_patterns_remove(struct platform_device *pdev)
93  {
94      // Get the led patterns's private data from the platform device.
95      struct led_patterns_dev *priv = platform_get_drvdata(pdev);
96
97      // Disable software-control mode, just for kicks.
98      iowrite32(0, priv->hps_led_control);
99
100     pr_info("led_patterns_remove successful\n");
101
102     return 0;
103 }
```

Listing 8: More useful remove function. The function gets the state container from the platform device, then puts the component into hardware-control mode. All memory that was allocated and mapped in our probe function is automatically cleaned up by the kernel because we used managed devres functions.

**To put this all together, update your driver as follows:**

- Add the following headers to your file:
  - `#include <linux/io.h>`: `iowrite32/ioread32` functions
- Add the code from listing 6 to the top of your file, below the include statements.
- Add the probe and remove functions from listings 7 and 8; these need to be above your `static struct platform_driver led_patterns_driver` definition, unless you use function prototypes.
- Define your register offsets (in bytes!) using `#define` statements at the top of your code. These offsets are used in lines 67-69 in listing 7.

**Verification**

Compile the driver and copy it `/srv/nfs/de10nano/ubuntu-rootfs/home/soc`. Loading your driver should turn all 8 LEDs on, and removing it should enable hardware-control mode!

### Miscdevice Creation

We're making progress! We've now turned LEDs on in the most complicated way yet! But wouldn't it be nice to control the LEDs outside of the probe function? To do this, we'll create a *character device* using the misc subsystem. This subsystem is just a light wrapper around the character device API, but it saves a lot of headache and boilerplate.

First, we need to add a `miscdevice` to our state container, as shown in listing 9. By putting the `miscdevice` in our state container, each LED patterns component in our system will get it's own `miscdevice`; although we only have one LED patterns, designing our driver to handle multiple LED patterns components is the best practice. Listing 9 also contains a *mutex*, but we can ignore that until we implement the character device's write function.

```
21  * @led_reg: Pointer to the led_reg register
22  * @miscdev: miscdevice used to create a character device
23  * @lock: mutex used to prevent concurrent writes to memory
24  *
25  * An led_patterns_dev struct gets created for each led patterns component.
26  */
27 struct led_patterns_dev {
28     void __iomem *base_addr;
29     void __iomem *hps_led_control;
30     void __iomem *base_period;
31     void __iomem *led_reg;
32     struct miscdevice miscdev;
33     struct mutex lock;
34 };
```

Listing 9: Updated LED patterns state container. `miscdev` will contain the `miscdevice`. `struct mutex lock` will be used to ensure that concurrent writes to the LED patterns happen in order (i.e., without interruption).

Once we have the `miscdev` in our state container, we can initialize it in our probe function, as highlighted in listing 10. The fields we initialize include the minor number, which we set to be dynamically allocated.

Devices have **major** and **minor** numbers. The major ID or number identifies the general class of the device, which is used by the kernel to lookup the appropriate driver for this type of device. The minor number identifies a particular device within the class.

Other fields include the character device name that will show up in `/dev/`; the file operations supported by the device; and the device's parent, which in our case is the platform device (the "physical" device, so to speak). See the `miscdevice` definition to see all of the fields. Once the device is initialized, all we need to do is register it with the misc subsystem using `misc_register()`.

All we need to do in the remove function is deregister the `miscdev`, as shown in listing 11.

Using the code in listings 10 and 11, the misc subsystem will create a character device for us at `/dev/led_patterns`. To actually use the device, we need to define what file operations are supported.

```
213 // Initialize the misc device parameters
214 priv->miscdev.minor = MISC_DYNAMIC_MINOR;
215 priv->miscdev.name = "led_patterns";
216 priv->miscdev.fops = &led_patterns_fops;
217 priv->miscdev.parent = &pdev->dev;
218
219 // Register the misc device; this creates a char dev at /dev/led_patterns
220 ret = misc_register(&priv->miscdev);
221 if (ret) {
222     pr_err("Failed to register misc device");
223     return ret;
224 }
225
226 /*
227  * Attach the led patterns's private data to the platform device's struct.
228  * This is so we can access our state container in the other functions.
229  */
230 platform_set_drvdata(pdev, priv);
```

Listing 10: How to initialize and register the `miscdev`. The `fops` field is a pointer to our `file_operations` **`struct`**, and the `parent` field is a pointer to the device **`struct`** contained within our `platform_device` **`struct`**.

`ret` **needs to be declared as a `size_t` at the top of the probe function.**

```
252 // Deregister the misc device and remove the /dev/led_patterns file.
253 misc_deregister(&priv->miscdev);
254
255 pr_info("led_patterns_remove successful\n");
```

Listing 11: How to deregister the `miscdev` in our `remove` function.

### Read/Write Functions

The `file_operations` **`struct`** is used to define what file operations are supported by a character device. The main operations we need are `open`, `close`, `read`, and `write`. The misc subsystem takes care of `open` and `close` for us, so we just have to define the `read` and `write` functions. Listing 12 shows the file operations structure we need to create.

```
146 /**
147  *  led_patterns_fops - File operations supported by the
148  *                      led_patterns driver
149  * @owner: The led_patterns driver owns the file operations; this
150  *         ensures that the driver can't be removed while the
151  *         character device is still in use.
152  * @read: The read function.
153  * @write: The write function.
154  * @llseek: We use the kernel's default_llseek() function; this allows
155  *          users to change what position they are writing/reading to/from.
156  */
157 static const struct file_operations  led_patterns_fops = {
158     .owner = THIS_MODULE,
159     .read = led_patterns_read,
160     .write = led_patterns_write,
161     .llseek = default_llseek,
162 };
```

Listing 12: LED patterns `file_operations` **`struct`**. Setting the `owner` field to to `THIS_MODULE` prevents our LED patterns module from being unloaded while its operations are in use. The `read` and `write` fields are set as pointers to our read and write functions.

We'll start with the `read` function shown in listing 13, as it's the simpler of the two. The comments in listing 13 explain what the function does.

The trickiest part is the `container_of` macro. Essentially, `container_of` finds the **`struct led_patterns_dev`** that *contains* the `miscdev` pointed to by `file->private_data` (remember that our state container conatins a `miscdev`); the misc subsystem assigns our `miscdev` to `file->private_data`.

Lines 63–75 handle some basic error checking. `copy_to_user()` handles copying the data from kernel space to user space. For numerous reasons, the kernel can't directly dereference user-space pointers; see this discussion in LDD3 to learn more. When a program reads from a character device, it will read until end-of-file (EOF) is reached; EOF is signified by returning a 0 in our `read` function. However, if we only return 0, the user-space program won't properly receive the data it tried to read. We need to return the number of bytes that were read and advance the file `offset` pointer by that number of bytes. We keep doing this

```c
/**
 * led_patterns_read() - Read method for the led_patterns char device
 * @file: Pointer to the char device file struct.
 * @buf: User-space buffer to read the value into.
 * @count: The number of bytes being requested.
 * @offset: The byte offset in the file being read from.
 *
 * Return: On success, the number of bytes written is returned and the
 * offset @offset is advanced by this number. On error, a negative error
 * value is returned.
 */
static ssize_t led_patterns_read(struct file *file, char __user *buf,
    size_t count, loff_t *offset)
{
    size_t ret;
    u32 val;

    /*
     * Get the device's private data from the file struct's private_data
     * field. The private_data field is equal to the miscdev field in the
     * led_patterns_dev struct. container_of returns the
     * led_patterns_dev struct that contains the miscdev in private_data.
     */
    struct led_patterns_dev *priv = container_of(file->private_data,
                                struct led_patterns_dev, miscdev);

    // Check file offset to make sure we are reading from a valid location.
    if (*offset < 0) {
        // We can't read from a negative file position.
        return -EINVAL;
    }
    if (*offset >= SPAN) {
        // We can't read from a position past the end of our device.
        return 0;
    }
    if ((*offset % 0x4) != 0) {
        // Prevent unaligned access.
        pr_warn("led_patterns_read: unaligned access\n");
        return -EFAULT;
    }

    val = ioread32(priv->base_addr + *offset);

    // Copy the value to userspace.
    ret = copy_to_user(buf, &val, sizeof(val));
    if (ret == sizeof(val)) {
        pr_warn("led_patterns_read: nothing copied\n");
        return -EFAULT;
    }

    // Increment the file offset by the number of bytes we read.
    *offset = *offset + sizeof(val);

    return sizeof(val);
}
```

Listing 13: Character device read function. The read function gets our state container, reads the requested value from hardware, then writes the value back to user-space.

until there are no bytes left to read, at which point we return `0` to signal EOF has been reached.

The operations we have to do in our `write` function are conceptually similar to the `read` function. The new concept in our write function is that we use a *mutex* to protect against race conditions. Mutex is short for *mutual exclusion*, which essentially means that only one thread/process can access a shared resource at any time. See the mutual exclusion Wikipedia page and the kernel's mutex documentation for more information. Essentially, we don't want two processes trying to write to the same register at the same time.

```c
/**
 * led_patterns_write() - Write method for the led_patterns char device
 * @file: Pointer to the char device file struct.
 * @buf: User-space buffer to read the value from.
 * @count: The number of bytes being written.
 * @offset: The byte offset in the file being written to.
 *
 * Return: On success, the number of bytes written is returned and the
 * offset @offset is advanced by this number. On error, a negative error
 * value is returned.
 */
static ssize_t led_patterns_write(struct file *file, const char __user *buf,
    size_t count, loff_t *offset)
{
    size_t ret;
    u32 val;

    struct led_patterns_dev *priv = container_of(file->private_data,
                                struct led_patterns_dev, miscdev);

    if (*offset < 0) {
        return -EINVAL;
    }
    if (*offset >= SPAN) {
        return 0;
    }
    if ((*offset % 0x4) != 0) {
        pr_warn("led_patterns_write: unaligned access\n");
        return -EFAULT;
    }

    mutex_lock(&priv->lock);

    // Get the value from userspace.
    ret = copy_from_user(&val, buf, sizeof(val));
    if (ret != sizeof(val)) {
        iowrite32(val, priv->base_addr + *offset);

        // Increment the file offset by the number of bytes we wrote.
        *offset = *offset + sizeof(val);

        // Return the number of bytes we wrote.
        ret = sizeof(val);
    }
    else {
        pr_warn("led_patterns_write: nothing copied from user space\n");
        ret = -EFAULT;
    }

    mutex_unlock(&priv->lock);
    return ret;
}
```

Listing 14: Character device write function. A mutex is used to prevent multiple writes to /dev/led_patterns from being reordered.

**We're now ready to implement our `miscdev`. Update your code as follows:**

1. Add the following headers:
    - `#include <linux/mutex.h>`: mutex defintions
    - `#include <linux/miscdevice.h>`: miscdevice defintions
    - `#include <linux/types.h>`: data types like `u32`, `u16`, `etc`.
    - `#include <linux/fs.h>`: `copy_to_user`, etc.
2. Update your state container according to listing 9.
3. Update your probe and remove functions according to listings 10 and 11.
4. Add the `fops` structure from listing 12 above your probe function.
5. Add the read and write functions from listings 13 and 14 above your `fops` structure.

### Verification

1. Compile the driver and copy it to `/srv/nfs/de10nano/ubuntu-rootfs/home/soc`.
2. Remove the previous module using `rmmod` if you haven't already, then load the new module. **You should see a character device show up at `/dev/led_patterns` on the SoC.**
3. Cross-compile the `led_patterns_miscdev_test.c` file, located in `sw/led-patterns/`.
4. Copy the `led_patterns_miscdev_test` executable to `/srv/nfs/de10nano/ubuntu-rootfs/home/soc/`.
5. Run `led_patterns_miscdev_test` in PuTTY. The program should read register values, display some values on the LEDs, and change the base period of your LED patterns component. 🎉

### Demo

Modify the `led_patterns_miscdev_test` program to recreate your custom pattern (from lab 4) on the LEDs. The program should display your custom pattern indefinitely (until you exit the program with ctrl+c). Demonstrate that the LED pattern is the same by showing the pattern in hardware-control mode and software-control mode (the hardware-control mode demonstration does not have to be included in your C program).

## Exporting Device Attributes with sysfs

Writing to our LED patterns registers via the character device is great, but a lot of times we just want to use simple command line tools like `echo` and `cat` instead of writing a C program. To do this, we need to create a *device attribute* for each register and export them via sysfs. For an overview of sysfs and attributes, see the kernel's sysfs documentation; the kernel's device documentation also contains some information about device attributes. **Read these documentation pages before moving on.**

In short, **sysfs** is a RAM-based filesystem used to export kernel data structures and their attributes to userspace.

### Creating Device Attributes

To simplify the creation of our device attributes, we'll use the `DEVICE_ATTR_RW()` macro; doing so removes most of the boilerplate required to define a read/write attribute. Once we've created a device attribute, we add it to an attribute group so the kernel can export the attribute for us. See the `ATTRIBUTE_GROUPS` macro to for some details on how it creates the attribute group for us. This post by Greg Kroah-Hartman (one of the main Linux kernel maintainers) provides details on how to properly create sysfs files, but the post is a little bit dated and doesn't reflect the interfaces/macros we are using; the post can be regarded as a description of how lower-level kernel code creates sysfs files for us. Listing 15 shows the code we use to create our device attributes.

```
325 // Define sysfs attributes
326 static DEVICE_ATTR_RW(hps_led_control);
327 static DEVICE_ATTR_RW(base_period);
328 static DEVICE_ATTR_RW(led_reg);
329
330 // Create an attribute group so the device core can
331 // export the attributes for us.
332 static struct attribute *led_patterns_attrs[] = {
333     &dev_attr_hps_led_control.attr,
334     &dev_attr_base_period.attr,
335     &dev_attr_led_reg.attr,
336     NULL,
337 };
338 ATTRIBUTE_GROUPS(led_patterns);
```

Listing 15: How to create read/write device attributes. The `DEVICE_ATTR_RW` macro creates the `device_attribute` **`struct`** for us, resulting in a variable called `dev_attr_<attribute-name>`. We then create an attribute group from a list of attributes. The `ATTRIBUTE_GROUPS()` macro expands `led_patterns` into `led_patterns_attrs` (which is the name of our attribute list).

Once we've created the attributes, we need to define its associated `show` and `store` functions. We'll start with the `led_reg` `show` function, shown in listing 16. The `show` function is responsible for returning the attribute as an ascii text-file; it gets called when anybody in user-space tries to read the attribute. Since our value is a `u8`, we use `scnprintf` to format the value as a string and place it into the user-space buffer. See printk-specifiers for information on the format string used in `scnprintf`.

The `store` method shown in listing 17 is called whenever someone in user-space writes to the attribute. We use `kstrtou8` to parse the string we were given into a `u8`; this function supports conversion from decimal-, hex-, and octal-formatted strings, and it ensures that the parsed value fits into an unsigned 8-bit integer.

Listing 18 shows the `show` and `store` functions for the `hps_led_control` register, and listing 19 shows the `show` and `store` functions for the `base_period` register.

```
272 /**
273  * led_reg_show() - Return the led_reg value to user-space via sysfs.
274  * @dev: Device structure for the led_patterns component. This
275  *       device struct is embedded in the led_patterns' platform
276  *       device struct.
277  * @attr: Unused.
278  * @buf: Buffer that gets returned to user-space.
279  *
280  * Return: The number of bytes read.
281  */
282 static ssize_t led_reg_show(struct device *dev,
283     struct device_attribute *attr, char *buf)
284 {
285     u8 led_reg;
286     struct led_patterns_dev *priv = dev_get_drvdata(dev);
287
288     led_reg = ioread32(priv->led_reg);
289
290     return scnprintf(buf, PAGE_SIZE, "%u\n", led_reg);
291 }
```

Listing 16: sysfs show method. This method gets our state container out of device associated with the attribute being read. It then reads the value from hardware, and fills the user-space buffer with a string that contains the led_reg value.

```c
/**
 * led_reg_store() - Store the led_reg value.
 * @dev: Device structure for the led_patterns component. This
 *       device struct is embedded in the led_patterns' platform
 *       device struct.
 * @attr: Unused.
 * @buf: Buffer that contains the led_reg value being written.
 * @size: The number of bytes being written.
 *
 * Return: The number of bytes stored.
 */
static ssize_t led_reg_store(struct device *dev,
    struct device_attribute *attr, const char *buf, size_t size)
{
    u8 led_reg;
    int ret;
    struct led_patterns_dev *priv = dev_get_drvdata(dev);

    // Parse the string we received as a u8
    // See https://elixir.bootlin.com/linux/latest/source/lib/kstrtox.c#L289
    ret = kstrtou8(buf, 0, &led_reg);
    if (ret < 0) {
        return ret;
    }

    iowrite32(led_reg, priv->led_reg);

    // Write was successful, so we return the number of bytes we wrote.
    return size;
}
```

Listing 17: sysfs store method. This is similar to show method in listing 16, except it receives and parses the string written by the user, then stores it in `priv->led_reg`. We don't need a mutex this time because the kernel layer above sysfs handles the mutex's for us; see this stackoverflow post for details.

```c
/**
 * hps_led_control_show() - Return the hps_led_control value
 *                          to user-space via sysfs.
 * @dev: Device structure for the led_patterns component. This
 *       device struct is embedded in the led_patterns' device struct.
 * @attr: Unused.
 * @buf: Buffer that gets returned to user-space.
 *
 * Return: The number of bytes read.
 */
static ssize_t hps_led_control_show(struct device *dev,
    struct device_attribute *attr, char *buf)
{
    bool hps_control;

    // Get the private led_patterns data out of the dev struct
    struct led_patterns_dev *priv = dev_get_drvdata(dev);

    hps_control = ioread32(priv->hps_led_control);

    return scnprintf(buf, PAGE_SIZE, "%u\n", hps_control);
}

/**
 * hps_led_control_store() - Store the hps_led_control value.
 * @dev: Device structure for the led_patterns component. This
 *       device struct is embedded in the led_patterns'
 *       platform device struct.
 * @attr: Unused.
 * @buf: Buffer that contains the hps_led_control value being written.
 * @size: The number of bytes being written.
 *
 * Return: The number of bytes stored.
 */
static ssize_t hps_led_control_store(struct device *dev,
    struct device_attribute *attr, const char *buf, size_t size)
{
    bool hps_control;
    int ret;
    struct led_patterns_dev *priv = dev_get_drvdata(dev);

    // Parse the string we received as a bool
    // See https://elixir.bootlin.com/linux/latest/source/lib/kstrtox.c#L289
    ret = kstrtobool(buf, &hps_control);
    if (ret < 0) {
        // kstrtobool returned an error
        return ret;
    }

    iowrite32(hps_control, priv->hps_led_control);

    // Write was successful, so we return the number of bytes we wrote.
    return size;
}
```

Listing 18: show and store methods for hps_led_control. Note the use of kstrtobool.

```c
/**
 * base_period_show() - Return the base_period value to user-space via sysfs.
 * @dev: Device structure for the led_patterns component. This
 *       device struct is embedded in the led_patterns' platform
 *       device struct.
 * @attr: Unused.
 * @buf: Buffer that gets returned to user-space.
 *
 * Return: The number of bytes read.
 */
static ssize_t base_period_show(struct device *dev,
    struct device_attribute *attr, char *buf)
{
    u8 base_period;
    struct led_patterns_dev *priv = dev_get_drvdata(dev);

    base_period = ioread32(priv->base_period);

    return scnprintf(buf, PAGE_SIZE, "%u\n", base_period);
}

/**
 * base_period_store() - Store the base_period value.
 * @dev: Device structure for the led_patterns component. This
 *       device struct is embedded in the led_patterns' platform
 *       device struct.
 * @attr: Unused.
 * @buf: Buffer that contains the base_period value being written.
 * @size: The number of bytes being written.
 *
 * Return: The number of bytes stored.
 */
static ssize_t base_period_store(struct device *dev,
    struct device_attribute *attr, const char *buf, size_t size)
{
    u8 base_period;
    int ret;
    struct led_patterns_dev *priv = dev_get_drvdata(dev);

    // Parse the string we received as a u8
    // See https://elixir.bootlin.com/linux/latest/source/lib/kstrtox.c#L289
    ret = kstrtou8(buf, 0, &base_period);
    if (ret < 0) {
        // kstrtou8 returned an error
        return ret;
    }

    iowrite32(base_period, priv->base_period);

    // Write was successful, so we return the number of bytes we wrote.
    return size;
}
```

Listing 19: `show` and `store` methods for `base_period`.

### Attaching Device Attributes to the Platform Device

With the show and store functions defined, we're almost ready to use our attributes. We just need to attach the attribute group to our platform driver so the driver core can create and export the attributes without race conditions. Listing 20 shows how to do this.

```
458 static struct platform_driver led_patterns_driver = {
459     .probe = led_patterns_probe,
460     .remove = led_patterns_remove,
461     .driver = {
462         .owner = THIS_MODULE,
463         .name = "led_patterns",
464         .of_match_table = led_patterns_of_match,
465         .dev_groups = led_patterns_groups,
466     },
467 };
```

Listing 20: How to add our attribute group to our platform driver, using the `dev_groups` field. See the device_driver struct documentation for details.

Now we're ready to test the attributes. **Update your driver as follows**:

1. Add `#include <linux/kstrtox.h>`: `kstrtou8`, etc. This header is already included in another header we're using, but being explicit doesn't hurt.
2. Add the code in listings 15, 16, 17, 18, and 19 to your driver; the `show` and `store` functions need to declared before you create the attributes (listing 15).
3. Modify your `led_patterns_driver` **`struct`** according to listing 20

#### Verification

1. Compile the driver and copy it to `/srv/nfs/de10nano/ubuntu-rootfs/home/soc`.
2. Remove the previous module using `rmmod` if you haven't already, then load the new module. If everything worked correctly, our attribute files should be located at `/sys/devices/platform/<base-address>.led_patterns` where `<base-address>` is the physical address of your LED patterns component.
3. `cd` to that directory and write/read values from/to the attributes. **You will need to be root to do this; run `sudo -i` to get a root shell.**

If all goes well, you should be able to control your LED patterns component! 🎉 Using sysfs attributes from the command line is very convenient for testing and debugging.

The `kstrtobool` function does not accept hex values! It accepts "YyTt1NnFf0" or on/off. See the kstrtobool documentation for more info. This means you can't write hex values to `hps_led_control`.

Phew! There are a lot of ways to write to our LEDs!
Here are locations that the led patterns device and associated driver show up in Linux (Your address is likely to be different depending on how Platform Designer assigned the base address.):

- `/dev/led_patterns` (raw bytes)
- `/sys/devices/platform/ff200020.led_patterns` (sysfs device attribute tied to the platform_driver struct)
- `/sys/bus/platform/devices/ff200020.led_patterns` (basically the same as the directory above, but just in a different place)

Information about the driver itself is in

- `/sys/bus/platform/drivers/led_patterns` This directory has symlink(s) to the device(s) that are bound to it (/sys/devices/platform/...) It also has a symlink to kernel module information (/sys/module/led_patterns).
- `/sys/` has a lot of information, and a lot of symlinks between the different parts of the tree (e.g. bus, devices, class, etc.). For example, the led_patterns character device shows up as a symlink in /sys/dev/char/, which links to /sys/devices/virtual/misc/led_patterns.
- `/proc/device-tree` This shows the live device tree.

## LED Patterns Revisited

All we ever do is light up LEDs, so let's light 'em up again! 💡🕺 For fun, we're going to create LED patterns again, but this time we'll use our device driver and *shell scripts*!

We'll learn by example. Listing 21 shows an example that creates an "alternating" pattern. Listing 22 left-shifts an arbitrary `LED_VAL`. See this bash cheatsheet for more details on bash syntax.

Both scripts start with `#!/bin/bash`. This is known as a "shebang". It tells the interpreter that called the script (bash, in our case) which interpreter to use when executing the rest of the script. This may seem a little weird when we are using a bash terminal to call a bash script, but we can also use shebangs to execute python scripts, perl scripts, etc. The main upside of using a shebang is that it allows us to execute the script using `./`, e.g. `./led_pattern0.sh`, instead of `bash led_pattern0.sh`; in order to execute the script this way, the script file needs to have execute permissions.

In bash, you can't have spaces around the = sign!

```
#!/bin/bash
HPS_LED_CONTROL="/sys/devices/platform/ff200020.led_patterns/hps_led_control"
BASE_PERIOD="/sys/devices/platform/ff200020.led_patterns/base_period"
LED_REG="/sys/devices/platform/ff200020.led_patterns/led_reg"

# Enable software-control mode
echo 1 > $HPS_LED_CONTROL

while true
do
    echo 0x55 > $LED_REG
    sleep 0.25
    echo 0xaa > $LED_REG
    sleep 0.25
done
```

Listing 21: Shell script to create an "alternating" LED pattern. `sleep` takes arguments in *seconds*. The paths to your attributes might differ from what's shown.

```
#!/bin/bash
HPS_LED_CONTROL="/sys/devices/platform/ff200020.led_patterns/hps_led_control"
BASE_PERIOD="/sys/devices/platform/ff200020.led_patterns/base_period"
LED_REG="/sys/devices/platform/ff200020.led_patterns/led_reg"

echo 1 > $HPS_LED_CONTROL

led_val=0x53;

while true
do
    echo $led_val > $LED_REG
    sleep 0.15
    # left-shift led_val; wrap the value when it overflows 0xff
    led_val=$(((led_val << 1) % 0xff))
done
```

Listing 22: Shell script to left-shift an arbitrary value on the LEDs. Anything inside **`$(())`** is treated as a math expression.

### Verification

**Test out the scripts in listings 21 and 22:**

1. Put the code in listing 21 into `sw/led-patterns/led_pattern0.sh`
2. Put the code in listing 22 into a file called `sw/led-patterns/led_pattern1.sh`
3. Copy the shell files over to your DE10 Nano's filesystem.
4. In PuTTY, give the shell scripts execute permissions with `chmod +x`
5. In PuTTY, execute the scripts, e.g. `./led_pattern0.sh`; the scripts are infinite loops, so you will need to use ctrl+c to stop the script.

# Deliverables

## Demonstrations

1. Demonstrate your modified miscdev test program. Show that your program implements the same custom LED pattern that your hardware state machine implements.
2. Manually control your LED patterns component from the command line by writing to the sysfs attribute files.
3. Write a bash script that demonstrates an LED pattern in software-control mode and changing the base period in hardware-control mode.

## GitHub Submission

Follow the workflow given in your repository (`docs/workflow.md`), per usual. Make sure all your files are committed.

### Questions

Answer the following questions in your markdown file:

1. What is the purpose of platform bus?
2. Why is the device driver's compatible property important?
3. What does the `probe` function do?
4. How does your driver know what memory addresses are associated with your device?
5. What are the two ways we can write to our device's registers? In other words, what subsystems do we use to write to our registers?
6. What is the purpose of our `struct led_patterns_dev` *state container*?